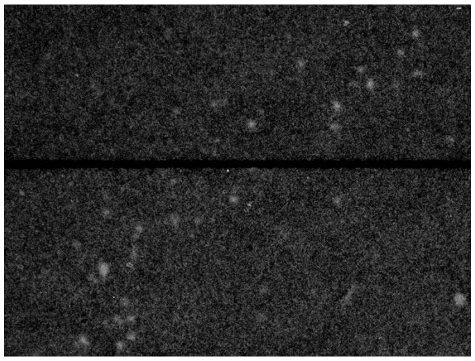
AD-A064 794 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
ERROR RECOVERY IN CAPABILITY SYSTEMS,(U)
JUN 78 W A WULF, D LANCIAUX F44620-73-C-0074
UNCLASSIFIED CMU-CS-78-127 AFOSR-TR-79-0061 NL

| OF |
ADA
064794

END
DATE
FILMED
4-79
DDC



CMU-CS-78-127

Error recovery in capability systems

Didier Lanciaux (1)

William A. Wulf

Department of Computer Science
Carnegie Mellon University

June 1978

Abstract: Methodologies and checking techniques have been proposed to improve sofware reliability. It has also been argued that capability mechanisms are the natural support for these techniques because they enhance modular decomposition and information hiding.
However, there is a conflict between these observations; modular decomposition limits the possible recovery actions to the information that a module can access directly. Each module must rely upon the reliability of those that it uses.
This paper presents a mechanism which allows recovery to be managed at any level in the system while satisfying the information hiding principle. It is based on a save-restore mechanism. In addition, primitives to define consistent states in the system are provided by the Kernel.

(1): IRIA-Laboria, Domaine de Voluceau, 78150 Le Chesnay, France.

This work was supported by the Advanced Research Projects Agency of the Department of Defense under contract F44620-73-C-0074 (which is monitered by the Air Force Office of Scientific Research).

1. Introduction.

Approaches to producing reliable sofware generally propose that programs be designed so as to be small, simple, and understandable. These approaches rely upon information hiding such as suggested by modular decomposition [Pa72] or data abstraction [LZ74]. Interdependencies among program fragments are thereby minimized and accessible data is reduced to a minimum. Thus, software faults should be more easily avoided and errors should be confined. Yet, hardware failures can occur which circumvent controls performed at compile time and error recovery is needed.

Error recovery requires: 1) that failures be detected, and 2) that the damage they may cause be limited as much as possible. Then, if possible, failures should be made transparent to higher levels of the system.

Capability-based systems have been shown to be a dynamic support to the above programming principles and to meet the recovery requirements [Li76, De76], particularily when type extension is employed. An extended-type object is an instance of a new type implemented by a module in terms of previously defined types. An object of a given type can only be operated by the appropriate type module; conversely, a module can only operate on objects it receives as parameters. Thus, the capability mechanism helps to enforce modularity and localize detection. Furthermore, since modules are independent and protected from each other, the possible damage that failures can cause is inherently limited.

Recovery requirements can be enforced by use of appropriate programming discipline [Wu75] and techniques [Po78]. Each module is responsible for the integrity of the abstract concept it implements. As a module "sees" only a module above (its caller) and possibly the modules below (its callee's), its reliability requirements are clearly defined. Parameter checking protects a module from misuse by its caller. Additional consistency tests are performed by the module on the objects it is responsible for. They are intended to detect possible latent failures as well as to verify that functions performed according to their specifications. If such checks failed, either the responsibility of repairing the erroneous information is taken by the module, or the concerned object is returned to a consistent state and the failure is reported to the levels above (in terms of the abstraction implemented by the module). Thus, a module must also be ready to handle reported errors from lower levels.

Detection as well as recovery rely upon either temporal or spatial redundancy. Checking that a function performed according to its specifications will require, for instance, that this function be partly performed again. On the other hand, internal consistency checks require the presence of redundant information against which tests can be performed. Both forms of redundancy are likely to be used together. Spatial redundancy can be provided in either of two ways:

> At the module level: When a module is activated on one of its functions, values of the objects it operates on may be saved so that returning them to a consistent state consists merely of restoring these values. The recursive cache provides such redundancy [Ra75]. When the acceptance test fails the previous consistent state is retrieved and either the operation can be corrected or an error can be raised to a higher level.
>
> At the object level: Mainly, redundancy is provided at this level in order to perform consistency checks on the object structure and to detect latent errors. For instance, double links in a list structure can provide for its repair.

The higher levels in the system generally provide less redundancy, since a loss there is often less vital to the system. This trade off is a matter for the implementation of the abstraction and does not necessarily reflects the importance that a user may attach to a given object.

This method is not wholly satisfatory. Object consistency relies on the implementation of the abstraction, and no control is left to its user. A user may be concerned with a sequence of operations on an object, and its consistency may be related to that whole sequence. Should an error be detected, the recovery action performed by the user may then require that the object be returned to the state it held before the sequence took place. Such recovery would require one to undo the sequence of operations. Even though the operations defined on a type may include their inverses, undoing the sequence requires recording the operations and the definition of a (possibly complicated) inverse sequence. This is likely to be source of new errors and may even be contradictory with other reliability requirements (e.g.,it may require rights on the object which otherwise would be denied). Moreover, a system crash during a sequence of operations can leave the objects inconsistent. This would be irrepairable since no record of the consistent state of these object has been kept. These remarks suggest that a global, user controlled approach is necessary.

2. The Object Model.

We noticed in section 1 that objects of extended-types are possibly formed of objects of other previously defined types. Hence, the overall structure of an object may describe a complicated graph when all components are considered. Moreover, this graph may evolve during the life of the object. Some objects may have their structure completely defined at their creation-time while other objects may add or remove components during their life time. Components may also be eventually shared among objects. According to the data abstraction principle, the structure of an object and its evolution are unknown at the level where it is used. As the evolution will probably be different for objects of the same type, it is not clear what copying an object means nor how to perform it. A model of object composition is required. We will consider two cases: the graph model and the component model.

In the graph model, each component of the object structure is a reference to an independent object and all are of equal significance. Copying, restoring or saving such an object is easy and concerns only the first level of the composition. Such a structure is exemplified by "directories" (Figure 2). A "directory" is an associative mechanism, mapping string names to object references. It is implemented as a capability-list which holds capabilities for objects, or for other directories which can be accessed independently or through other structures. The structure of the objects that the directory holds is of no concern; these objects are maintained at other levels of the system. Therefore copying a directory consists of only copying the capability-list.

In the component model, the representation of an object also holds the representation of its components; however,the components are an integral part of the object. An example of such a structure is a file (Figure 2). A file can be formed of a semaphore (used to implement the open-close operations) and a segment which contains the actual contents of the file. The semaphore and the segment are not independent entities; they are part of the representation of the file. Therefore, copying such an object also requires copying the representation of its components.

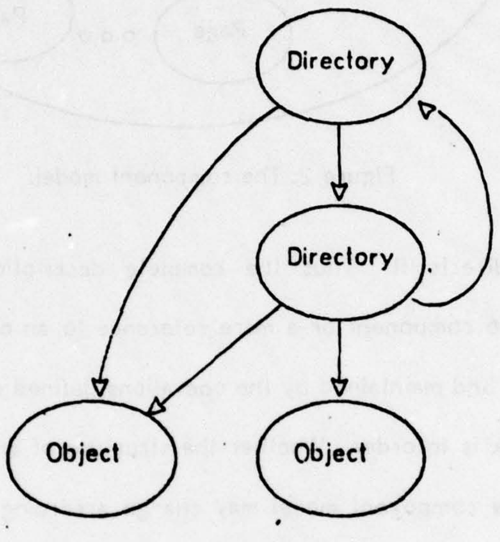


Figure 1: The graph model.

These two examples represent extremes of possible object structures. One is a pure graph model; the other is a pure component model. Even if examples can be found of these two basic models, real structures may make use of both of them at the same time. For instance, a semaphore may be viewed as holding a component, its value, and a reference to the first element of a process list. The processes on this list are not components of the semaphore. Furthermore, this structure evolves during the life of the object. Fortunately, the nature of this evolution is completely known by the

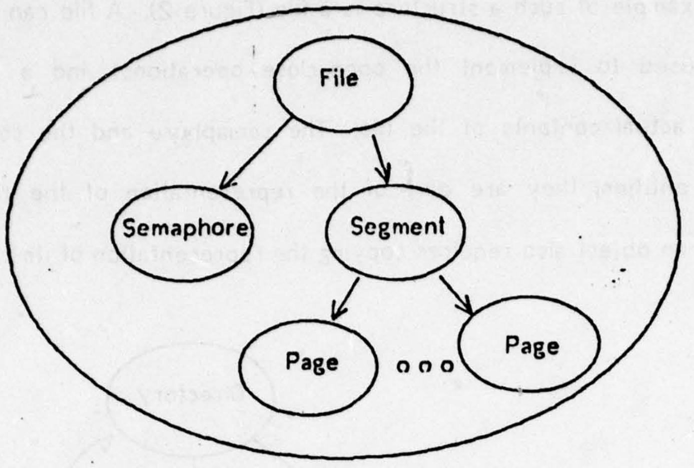


Figure 2: The component model.

operation which directs it. Thus the complete description of whether a given component is a true component or a mere reference to an object can be kept at the level of the object, and maintained by the operations defined on its type.

One more remark is in order. Whether the structure of an object is related to the graph model or the component model may change according to the semantics of the operation. Considering the file example again, it will be viewed by the file operations as matching the component model. However, the saving of the file will probably not involve the saving of the semaphore which therefore will no longer be considered as a "true" component.

Here we will consider components of a structure with regard to the saving of the state of an object. It is now clear that such operations are completely defined by the type of objects since the type operations are able to maintain a description of their structure.

3. Introduction to the mechanism.

So far we have considered local consistency at each level of abstraction. The introduction asserted the need for global consistency as well. We will distinguish between "vertical consistency" and "horizontal consistency". Roughly, vertical consistency refers to the sequential use of objects, and horizontal consistency refers to preserving some invariant relation between a set of objects.

3.1. Vertical consistency.

We previously noticed that the consistency of a sequence of operations should be considered. For local consistency it was possible to define a consistent state before an operation took place, and so it is also possible to consider the consistent state which existed before a sequence was performed. Recovery consists of restoring this state.

Since objects can be composed of several levels, the consistency of an object relies on the consistency of its components down to the primitive levels. Hence, the restoring of an object implies restoring all these components.

At the level where we consider it, the implementation of objects is hidden. The program therefore requires the assistance of a mechanism to define and save consistent states. To preserve "information hiding", this mechanism should keep the implementation of objects hidden, and should not require knowledge of the type of the object to be saved. It should be kept as independent of the object itself as possible, and possibly be provided by the kernel.

3.2. Horizontal Consistency.

A simple example will suffice to define horizontal consistency. Consider a data base which holds the status of the checking accounts in a bank. The following transaction records a transfer of 100 dollars from account A to account B:

```
read A;
A ← A - 100;
write A;
read B;
B ← B + 100;
write B;
```

The data base must be kept globally consistent; here this means that A+B must be the same before and after the transaction takes place. Suppose a failure occurs after A has been modified and that this failure requires B to be restored. B cannot be restored alone since A was already modified and the above requirement on A+B would no longer hold.

3.3. Introduction to the mechanism.

Global consistency requires ties between objects to be defined. However, in the case of horizontal consistency program assistance is needed. One may notice that this same problem occurs when the internal structure of an object is considered. At each level of the composition components are tied by the same sort of consistency relation.

To allow these relationships to be expressed, we introduce the concept of consistency-sets. Providing that α is a set formed of A and B, the transaction of our example would be written:

```
enter a;
    transaction body;
exit a;
```

A consistency-set is an object whose type is implemented by the kernel. The purpose of consistency-sets is to define the scope of recovery actions and the set of objects that are involved. Hence, access to these objects needs not be restricted. Once a set has been entered, its objects are accessed freely by normal use of capabilities.

Operations defined on this type are: enter, in, exit, restore. The enter and in operations are performed in order to declare the current state of objects in the set to be consistent with regard to the program. It triggers the saving of these objects. Hence, in our example, A and B would be considered together and therefore kept consistent with each other instead of being considered separately. The purpose of the exit operation is to mark the end of the use of the set. It tells also that the objects in the set are consistent again with regard to the transaction. Therefore, the restore operation, which allows a set to be returned to the (consistent) state prior to the enter, can only be performed if the set has not yet been exited. In order to allow the identification of objects' versions and the restoring of a set, both set and object versions are stamped with the current clock time when sets are entered.

4. Vertical Consistency.

Consistency-sets are intended to mark the consistent state of a collection of objects before an action takes place. This action will probably be expressed in terms of more elementary actions that are implemented by functions defined in lower level modules. Recovery requirements of these modules may lead to the declaration of consistency-sets which may intersect with those previously entered.

For instance, consider our previous data base example. For the purpose of recovery, the user declared the checking accounts A and B in the set a. Imagine now that the user called the data base manager in order to perform an operation on the checking account A. In order to manage its own recovery actions, the data base manager probably declares a new set *b* which includes A. As far as the data base manager is concerned, the state of A before it was called is consistent. If a recovery action is to be performed, this state must be restored. Consequently, when the new set *b* is entered, the state of A must be saved.

Now consider the transaction: when it receives control from the data base manager, the current state of A must be the state it held when the set *b* was exited. The same requirements apply to any consistency-sets that are entered sequentially. Therefore when an enter operation is performed on a set which intersects with a set that the same process already entered, new current versions are provided for any object of this set, whether current versions already exist or not. The purpose of the enter operation is to define a new "current" version. Consequently, between the enter and the exit of a set, several versions may have been stacked for some of its components. However, the restore operation requires the retrievial of the version of each object of the set from which the current version was originally created. In order to make these versions identifiable, sets are stamped with the current clock time when they are entered. When a new version is created for an object, this version is stamped in the same way. Hence, when restoring a set is required, the correct version of its objects can be found by matching the time stamp that the set holds with the time stamp of the different versions of its objects. The correct version of an object is the version which holds the time stamp whose value is the closest but smaller than the time stamp of the set. Figure 3 shows this process.

As the enter operation declares the current version of its objects to be consistent with regard to the coming transaction, so the exit operation declares the versions created for the purpose of this transaction to be consistent again. But once a set has been exited, new versions may be created through the same process for some of the objects it holds, so that the consistent state that it refers to would be lost. Therefore, when sets are exited, they are stamped with the current clock value. The versions of objects that such a set refers to are those whose stamps are the closest but smaller than its own stamp. However, more recent versions can be referred to in other sets and define a consistent state. The restoring of old versions would destroy these consistent states. Therefore, once a set is exited, the restore operation can no longer be performed. Instead new objects whose representations are copies of the referred versions can eventually be provided.

As a result of the exit operation, when intersecting sets are acted on sequentially, the use of objects contained in the embedding sets destroys the consistent state reffered to by inner sets. Figure 3.2 illustrates this remark, where the versions exited by the set S2 become the current versions of the set S1. If now objects of the set S1 are accessed, the state that set S2 refers to is no longer consistent with regard to the

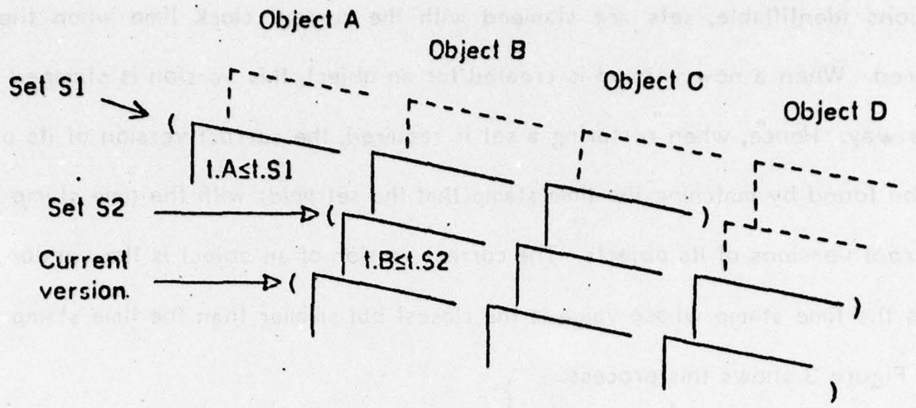


Figure 3.1: States of two nested sets.

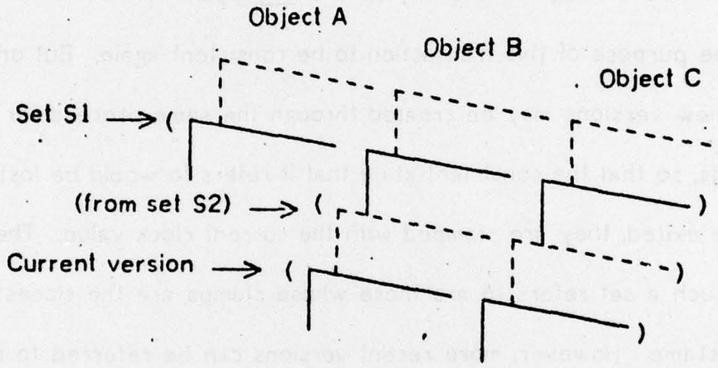


Figure 3.2: Exiting an inner set.

action performed through S2. The correct use of sets would require that further accesses to the objects of the set S1 be done by entering a new set. Accesses to these objects should never be done through S1 whose purpose is to refer to a previous consistent state.

Noticing that new versions are provided only when the first access is attempted, the objects that an inner set such as S2 holds can be put in such a state that any attempt to access them triggers the same process as when the outer set is entered for the first time. Hence, the creation of new versions would be provoked and the consistent state that the inner set holds would be saved.

A similar problem arises when the restoration of an outer set is made. The consistent states that inner sets refer to are destroyed. However, one may notice that the nesting of the sets reflects the nesting of operations which are to be cancelled when object restoring occurs at outer levels. Hence, such behavior is justified by the fact that the restored outer set had not yet been exited.

5. Horizontal Consistency.

Horizontal consistency refers to relations between objects. We extend it here to the case of objects used concurrently by parallel processes. As opposed to sequentially entered sets, objects which belong to sets used by concurrent processes must exhibit the same version. This leads to conflicts which are known as resulting in a "domino effect" [Ra75].

Considering our data base again, a transaction t1 may be currently updating the checking accounts A and B while a transaction t2 is currently updating the checking account B and C. Rules have been exhibited which allow t1 and t2 to execute concurrently and consistently [Gr75]. However, if some failure in transaction t1 results in the restoring of A, the data base consistency may require B to be restored at the same time by restoring the set S1={A,B}. As the restoring of B requires the restoring of C through the set S2={B,C} declared by the transaction t2, this leads eventually to a snow-ball effect where the transitive closure of all intersecting sets must be taken into account. Moreover, as the restoring of sets is not necessarily handled by the processes which hold them, it results in processes backing-up. This can lead to take into account even non intersecting sets.

Figure 4 exemplifies this where squares exhibit the relations between processes. Should process 4 fail at the point X, then the set of process 3 should be restored. But at the considered time process 3 was interacting with process 2 through intersecting sets, and so the same procedure must be repeated with process 2 and then with process 1. Issues related to recovery requirements have been analyzed in [Ra75].

Processes must cooperate in providing the necessary recovery structure (referred to as a "conversation"). While in our example transaction t1 was concerned with the

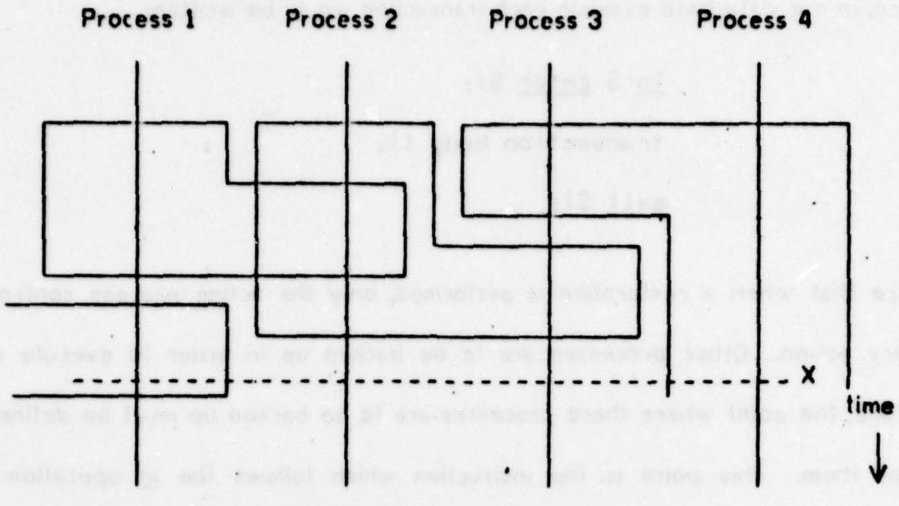


Figure 4: Conversations between concurrent processes.

set S1={A,B} and transaction t2 with S2={B,C}, the real recovery structure is formed of S=Union(S1,S2). The set S cannot be formed merely by merging S1 and S2; the involved processes must have agreed in the definition of the recovery structure and this structure must be common to the set of processes. Therefore, we will distinguish between conversations and usual recovery structures.

When an enter operation is attempted on a set which intersects with a set that another process is using, the entering of this set is refused and the operation fails. For this purpose, when a set is entered it is stamped with the process identification; such a stamp is removed when it is exited. Conversations require that the set which covers all the involved objects be previously entered. The entering of a conversation by a process declares at the same time the recovery point (exactly as the enter operation did for other recovery structures) and the conversation it refers to. For this purpose a new operation is introduced: in S enter Si where S is the covering set while Si is the subset used by the process. The set Si is linked to the set S and to its other subsets. The restore Si operation triggers the restoring of S.

Hence, in our data base example each transaction would be written:

```
in S enter Si;
transaction body ti;
exit Si;
```

Notice that when a restoration is performed, only the acting process controls its recovery action. Other processes are to be backed up in order to execute again. Therefore, the point where these processes are to be backed up must be defined for each of them. This point is the instruction which follows the in operation they executed. Such a back-up point can be saved in their subset when they execute the in operation.

The exit operation, as we previously defined it, declares a set to be consistent. In the case of a conversation such declaration must be extended to the covering set. Therefore the exit operation is really performed when all the subsets have been exited.

Finally, as mentioned earlier, the spreading of recovery actions may affect processes which were not cooperating through intersecting sets. The same remarks as above apply and we require all conversations to be nested so that recovery actions can be defined without side effects. Hence, once an in operation has been performed by a process any other in operation it performs must refer to a subset that it has already entered as a covering set.

6. Implementation.

The creation of a consistency-set takes a list of objects as an argument and returns a capability for the created set. When newly created, a consistency-set does not refer to any consistent state since it has not yet been stamped with a clock-time.

When a set is entered, the objects it contains should be saved. But notice that they are likely not to be in core. Therefore, instead of making a copy of them on secondary storage their representation is brought into core and declared as the new version, then space on secondary storage is provided for it. When the object representation is already in core, it is considered as the new one and is swapped out so as to update the previous version on secondary memory. Hence, creating a new version for an object only requires secondary storage space allocation.

Remember that a capability identifies a unique descriptor of each object, and that this descriptor contains the identification of the object representation (in terms of addresses). What the object descriptor refers to is the current representation. It is the head of a list of all the object versions. Thus, restoring the previous state of an object consists in substituting the necessary links. Providing that a tag field is present in the object descriptors for this purpose, the loading of object representations can be postponed until their real use. Hence, when a set is entered, the swapping of the objects' current versions on to the secondary storage is triggered by the kernel, and a load tag is set in the descriptors so as to provoke the creation of a new version when they are accessed for the first time. The creation of these new versions will provoke the descriptors to be stamped with the current clock-time so as to permit their identification.

According to the object model, when a new object version is created only its "true"

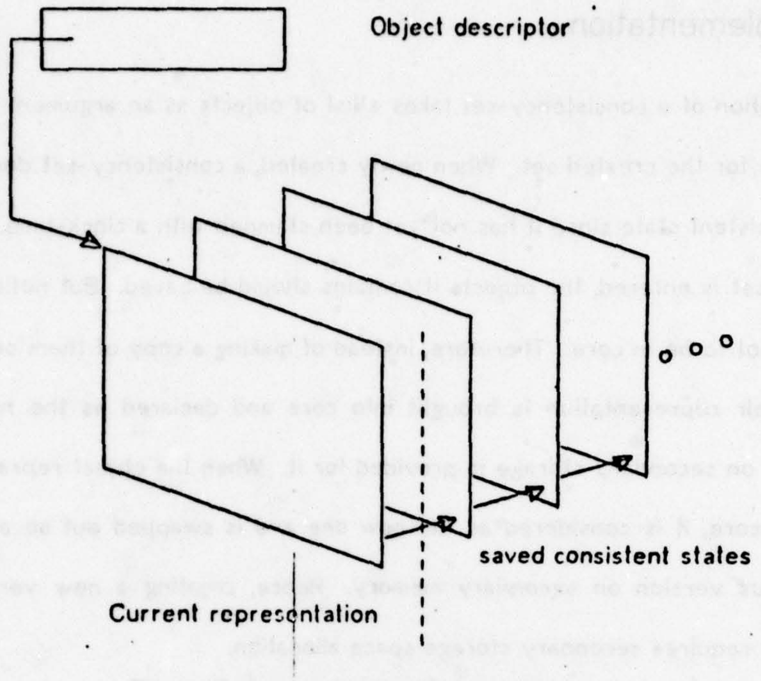


Figure 5: Saving the state of an object.

components need to be provided with a new current version. Only the first level of the object structure is considered; it is a capability-list. This capability-list contains the identification of its true components which themselves hold the identification of their own true components and so forth down to the primitive levels. These components can be described in the object structure by a consistency-set. The representations of the lower level components are not directly copied except when an access is required since we only need to keep track of the modifications. Providing that the load tag of an object is on, when the new version of this object is created, the kernel can then enter its consistency-set and trigger the creation of new versions for its components in the same way as for other objects. New versions will thus be created recursively for all components down to the primitive level. Hence, the only representations which are copied by the kernel are capability-lists, and when the lowest level of the structure is reached, segments.

In addition, object descriptors are provided with a process field that the enter operation checks. Recall that the entering of a set which intersects with an already entered set can only be made within the same process. Therefore when an enter operation is performed, each descriptor is stamped with the acting process identification. If the process field is already filled, its contents should match the executing process identification; otherwise it could only be entered by an in operation. The in operation only requires the declared subset to be included in the covering set; the subset is then linked to the covering set and to any other subsets that have been entered already.

When a set is exited, all the above information is removed. In addition the clock field of the set descriptor is updated so as to now refer to the most recent object versions. Therefore, the free use of these most recent versions without entering a set should be avoided since it would destroy a consistent state. For this purpose, when a set is exited, its object descriptors are initialized so as to trigger the creation of a new version when an attempt is made to access the corresponding objects. Multiple copies that such a mechanism might imply can be avoided easily by use of the modify tag usually provided in each object descriptor for the purpose of memory management. Hence, when a set is entered, the creation of new versions will not occur for objects whose representation was not altered.

Although it may look quite complicated at first glance, the management of consistency-sets is rather simple since it covers most of the usual actions of the virtual memory manager. However, it does use more space on secondary storage than a conventional virtual memory. We would like to point out that a garbage collection is any case necessary. Therefore, the secondary storage can be cleaned up by collecting the space of versions which are no longer referred to by a set. Each time a set is stamped with a clock time the previous value of its clock field defines object versions which are no longer usable. However, since the clock time of an object does not necessarily match the clock time of some set, care is needed. Given an object version, a count of the number of sets which refer to it should suffice to determine whether or not the space that this version uses can be collected.

7. Conclusion.

Current recovery techniques are limited by the fact that providing redundancy implies the access to the representation of objects. Therefore, only local redundancy can be provided, and recovery is either limited to errors detected at lower levels, or implies constructing audit trials so that actions can be undone. The resulting techniques involve careful design of the recovery actions and consequently are more likely to be reserved to the implementation of the lower levels of a system. Consistency-sets, on the other hand, seem to provide a robust and simple mechanism. By relying directly on the kernel of the system, they should also be more reliable than techniques which require more programming assistance.

Although it was beyond the scope of this paper to discuss the implementation of such a mechanism in detail, we have outlined it in order to show that the implied overhead should be small; providing a new version of an object representation usually requires only secondary storage space allocation. Moreover, explicit redundancy and consistency checks at the object level should be partly avoided since an old version can always be retrieved. A copy operation which can be implemented on the same basis as the creation of a new version for an object allows access to such information.

The main problem that such a mechanism raises is increased secondary storage requirements and the collecting of the memory space used by non-accessible object representations. The saving of space on secondary storage can be included with fail safe facilities where old versions would be periodically archived. However, the problem of retrieving non-usable versions of objects remains.

8. References.

De76	Denning, P.J., Fault Tolerant Operating Systems, *Computing Surveys*, (December 1976).
Gr75	Gray, J.N., R.A. Lorie, G.R. Putzolu, I.L. Traiger, Granularity Of Locks And Degrees Of Consistency In A Shared Data Base, *IBM Research Report*, RJ 1654, (September 1975).
Li76	Linden, T.A., Operating System Structures To Support Security And Reliable Software, *Computing Surveys*, (December 1976).
LZ74	Liskov, B.H., S. Zilles, Programming With Abstract Data Types, *SIGPLAN Notices*, (April 1974).
Pa72	Parnas, D.L., On The Criteria To Be Used In Decomposing Systems Into Modules, *CACM*, (December 1972).
Po78	Pollack, F.J., A Design Methodology For Fault Tolerant Software, *Ph.D. Thesis*, Carnegie Mellon University, (1978).
Ra75	Randell, B., System Structure For Software Fault Tolerance, *IEEE Transactions On Software Engineering*, (June 1975).
SS75	Saltzer, J.H., M.D. Schroeder, The Protection Of Information In Computer Systems, *Proceedings Of The IEEE*, (September 1975).
Wu75	Wulf, W.A., Reliable Hardware-Software Architecture, *SIGPLAN Notices*, 10, 6, (1975).

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 18 19 AFOSR-TR-79-0061 / 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 ERROR RECOVERY IN CAPABILITY SYSTEMS	5. TYPE OF REPORT & PERIOD COVERED Interim
	6. PERFORMING ORG. REPORT NUMBER 14 CMU-CS-78-127
7. AUTHOR(s) 10 William A. Wulf ~~and~~ Didier Lanciaux	8. CONTRACT OR GRANT NUMBER(s) 15 F44620-73-C-0074
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Department of Computer Sciences Pittsburgh, PA 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 16 ~~61101E~~ A02466/7 17 66
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agnecy 1400 Wilson Blvd. Arlington, VA 22209	12. REPORT DATE 11 June 1978
	13. NUMBER OF PAGES 24 12 27p.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Air Force Office of Scientific Research/NM Bolling AFB, Washington, DC 20332	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Methodologies and checking techniques have been proposed to improve software reliability. It has also been argued that capability mechanisms are the natural support for these techniques because they enhance modular decomposition and information hiding.

However, there is a conflict between these observations; modular decomposition limits the possible recovery actions to the information that a module can access directly. Each module must rely upon the reliability of those that it uses.

(Continued on back page)

DD FORM 1 JAN 73 1473

UNCLASSIFIED 403 081

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

20. Abstract continued.

This paper presents a mechanism which allows recovery to be managed at any level in this system while satisfying the information hiding principle. It is based on a save-restore mechanism. In addition, primitives to define consistent states in the system are provided by the Kernel.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)